



Flux CD

D1 Reference Architecture

Guide

Rationale and security considerations for the adoption of
Flux CD D1 reference architecture on Kubernetes

Table of Contents

Executive Summary	2
Introduction and Objective	3
Background	5
Organisation Suitability	5
User Personas	5
Low Level Design Description	8
Cluster Operating Model and Architecture	8
Example Workflows	14
Further Implementation Guidance	27
Securing GitHub	27
Multi-Tenancy Considerations	33
Other Security and Availability Considerations	38
About ControlPlane Enterprise for Flux CD	40
Appendix	42
Administrative Tasks	42
About	49
Team	49
Reviewers	49

Executive Summary

Flux CD, a cloud native Continuous Deployment tool, has emerged as a crucial component in modern GitOps pipelines. In this guide we delve into the intricacies of deploying a Flux CD reference architecture, offering a comprehensive understanding of the project's functionality, implementation, and best practices, to aid organisations seeking to harness the power of Flux CD for streamlined Continuous Deployment in cloud native environments.

It is mainly intended for those with some previous Flux knowledge, but whether you are a seasoned DevOps practitioner or a newcomer to the field, we will equip you with the knowledge and insights needed to succeed in your Flux CD journey.

We provide essential background information, detailing the prerequisites and suitability of Flux CD for various organisational setups. Through the delineation of user personas and roles, we elucidate how Flux CD fits into different team dynamics, ensuring a tailored approach to implementation.

The guide then delves into the low-level design description of Flux CD, elucidating the cluster operating model and architecture, and exploring the intricacies of the *d1-fleet*, *d1-infra*, and *d1-apps* repositories and their roles within the Flux ecosystem.

Drawing on real-world examples we outline various workflows, including the process of adding a new Application team and managing applications effectively, with implementation guidance addressing crucial aspects of GitOps such as GitHub as a production service, user access control, and authorization.

We further cover Flux CLI usage and alternatives, namespace RBAC considerations, and address potential security concerns, including tenant denial of service abuse and the role Kyverno policies play within Flux to help isolate tenants.

Finally, readers will find additional information about the subscription-based ControlPlane Enterprise for Flux, which supports the full-time employment of core maintainers to work on the upstream Flux CD project, and an appendix containing the most common administration tasks.

Introduction and Objective

In modern cloud native computing, efficiently and securely managing infrastructure and applications is crucial. Cloud native technologies have exposed the limitations of traditional infrastructure management methods in meeting the demands of agility, scalability, security and reliability.

GitOps embodies the principles of Git, the popular version control system, to automate the deployment, monitoring, and management of cloud environments. GitOps is used for operating modern cloud native infrastructure and applications, at its core, GitOps treats infrastructure as code, enabling teams to define, track, and version their infrastructure configurations alongside application code.

GitOps promotes a declarative approach to infrastructure management, where the desired state of the system is defined in code and stored in a Git repository. Continuous synchronisation between the desired state described in the Git repository and the actual state of the system is achieved through automated workflows and reconciliation mechanisms.

As organisations continue to embrace cloud native technologies and adopt DevOps practices, GitOps emerges as a foundational framework for managing cloud environments at scale. In this guide, we explore Flux, a leading GitOps tool, and its role in revolutionising cloud operations by streamlining the deployment, monitoring, and management of Kubernetes clusters and cloud native applications. We delve into the features, benefits, and best practices of Flux, illustrating how it enables organisations to embrace GitOps principles and unlock the full potential of cloud native ecosystems with improved security.

The purpose of this guide is to provide a comprehensive reference architecture demonstrating how Flux can address the needs of organisations with multiple teams looking to deploy applications on Kubernetes using GitOps principles. The guide aims to build upon the existing Flux documentation, showcasing how Flux can orchestrate multi-tenant clusters while catering to the different teams and stakeholders within an organisation.

We build upon the existing Flux documentation, which features a quickstart guide orchestrating a single cluster, to demonstrate how Flux can orchestrate large fleets of multi-tenant clusters with the D1 Reference Architecture.

This guide caters for the different teams and stakeholders within an organisation (i.e. platform, services and application teams), whilst maintaining scalability, operational efficiency and security. We focus on the main challenges organisations face when

adopting Flux CD in multi-team, multi-tenant environments, providing solutions to difficulties such as:

- Segregating responsibilities and access between platform, services, and application teams
- Maintaining security and compliance in a multi-tenant environment
- Ensuring scalability and operational efficiency in large-scale deployments
- Implementing best practices for GitOps workflows and repository structures

After completing this guide, readers will have a clear understanding of how to implement a Flux CD reference architecture that caters to the needs of their specific organisational structure. They will be equipped with the knowledge and best practices to:

- Set up and manage multi-tenant Kubernetes clusters using Flux CD
- Define and maintain a repository structure that supports collaboration between platform, services, and application teams
- Implement secure GitOps workflows and access control mechanisms

The reference architecture detailed within the next sections can be deployed with either Enterprise for Flux CD or the open source version of Flux, as both have feature parity.

This document is the first of a series of reference architectures designed to meet the complex needs of different organisational structures and ways of working.

Background

Organisation Suitability

Organisations that would be suitable for this particular reference architecture implementation are likely to meet some of the following criteria:

- Homogenisation of application packaging into containers
- Choice of Kubernetes as the orchestration platform
- A large number of clusters, or requirements to group applications resulting in multi-tenant, multi-cluster fleets
- *Namespace as a Service* operating model that involves a central Platform team responsible for Kubernetes cluster provisioning, fleet maintenance, and user experience, with application teams as cluster tenants
- Service teams responsible for maintaining and operating an individual service across a fleet of clusters (e.g. observability/security tools)
- GitHub (or equivalent) qualifies as a production-level service in terms of security and availability

User Personas

When operating Kubernetes clusters at scale, it is critical to define the responsibilities of those who are setting up the Kubernetes environment from those who use that Kubernetes environment. Within this document, we have named the former the *Platform team*, while the latter is referred to as the *Application team*.

Platform Team

<i>Characteristic</i>	<i>Description</i>
Goals	<ul style="list-style-type: none"> ● Provisioning and maintenance of multi-tenant Kubernetes clusters across the organisation ● Providing a tenant-friendly experience ● Adhering to SLA uptime and security requirements
Behaviours	<ul style="list-style-type: none"> ● Plans feature releases that are tested and trialled in non-production environments first (these may include feature deprecations i.e. Kubernetes API deprecations) ● Issues hotfix releases to all environments to address availability issues and bugs ● Maintains the same configuration in all clusters ● Uses cluster admin console/CLI access to troubleshoot issues
Motivations	<ul style="list-style-type: none"> ● Wants a simple way of managing clusters en masse ● Prefers minimal cluster-specific configuration ● Wants to eradicate cluster configuration drift
Pain Points	<ul style="list-style-type: none"> ● Application teams not configuring their applications in accordance with best practice ● Managing releases at scale, obtaining a feedback loop to understand when new releases break application workloads
Risk Behaviours	<ul style="list-style-type: none"> ● Will not apply security policy to own namespaces and will run privileged platform workloads without assessing for least privilege on the cluster

Application Team

<i>Characteristic</i>	<i>Description</i>
Goals	<ul style="list-style-type: none">● Building and running containerised applications on Kubernetes
Behaviours	<ul style="list-style-type: none">● Has different configurations between non-production and production environments (i.e. log levels, external endpoint addresses)● Requires multiple namespaces in non-production clusters● Aims to release as frequently as possible.
Motivations	<ul style="list-style-type: none">● A simple mechanism for consuming Kubernetes● Instant deployment feedback
Pain Points	<ul style="list-style-type: none">● May not be Kubernetes configuration experts● Kubernetes platform changes may cause breaking changes to their application
Risk Behaviours	<ul style="list-style-type: none">● Will configure their applications by any means possible to get them working, which without preventive controls could cause security misconfigurations or unusual configurations for the Platform team to support● Will turn off namespace level controls to get their applications working if required

Low Level Design Description

Cluster Operating Model and Architecture

D1 Architecture Repositories Structure

To enable the requirements and needs of the user personas described in the previous section, the next repository structure has been created:

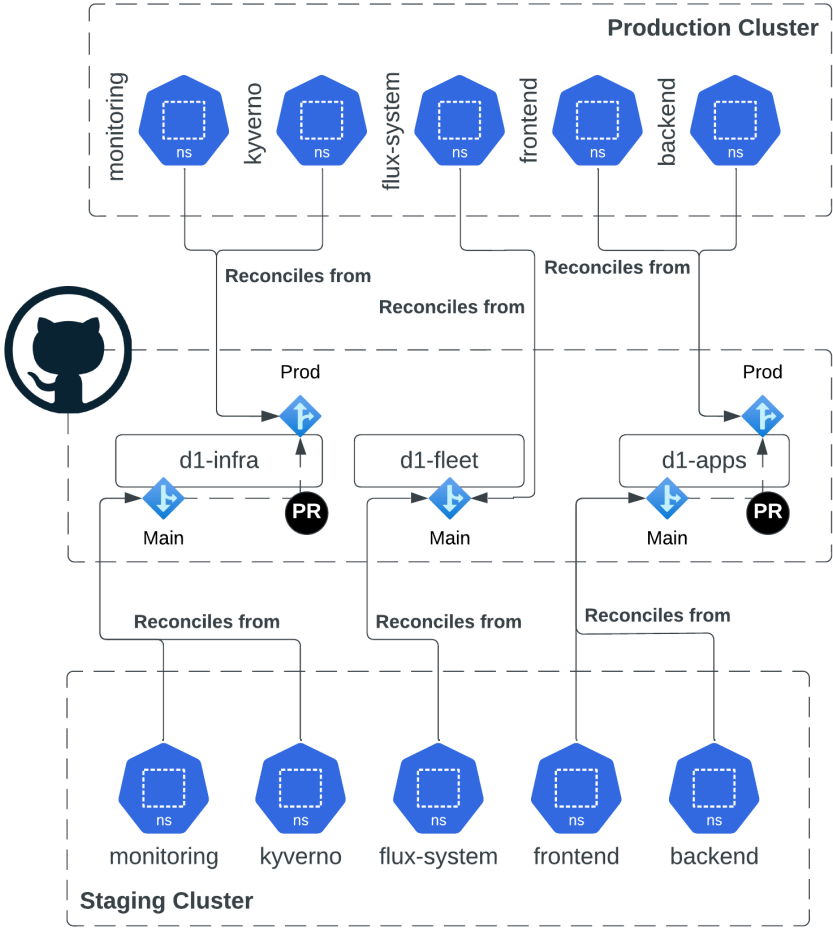


Figure 1: D1 Architecture - repository structure

d1-fleet Repository

Utilised by the Platforming team only, who are admins on all clusters in order to:

- Bootstrap Flux with multi-tenancy restrictions on fleet clusters
- Configure the delivery of platform components (defined in *d1-infra* repository)
- Configure the delivery of applications (defined in *d1-apps* repository)

d1-infra Repository

This repository is managed by the Platform team who are responsible for the Kubernetes infrastructure. This repository is used to define the Kubernetes infrastructure components such as:

- Cluster add-ons (CRD controllers, admission controllers, monitoring, logging, etc.)
- Cluster-wide definitions ([Namespace](#), [IngressClass](#), [StorageClass](#), etc.)
- Pod security standards
- Network policies

In larger organisations, where dedicated teams may be responsible for services (i.e. a security team managing policies, secrets team managing secrets operators, or observability team managing Prometheus), these teams will be configuring their components within this repository.

d1-apps Repository

Each Application team responsible for the delivery of an application running on the Kubernetes cluster fleet will be allocated an Application Repository, hosting application components such as:

- Flux [HelmRepository](#) CR pointing to the application Helm charts in container registries
- Flux [HelmRelease](#) CR for the applications with custom configuration per environment

This repository is reconciled on the cluster fleet by Flux as the namespace admin, and can't contain Kubernetes cluster-wide definitions such as CRDs, [ClusterRoles](#), [Namespaces](#), etc.

Access to this repository is restricted to the dev teams and the [Flux bot account](#). The Flux bot should be the only account with direct push access to the *main* branch.

Branching Strategy

The *main* branch of the *d1-fleet* repo controls all Flux instances, and the [Kustomization/GitRepository](#) pointers to the *d1-infra* and *d1-apps* repositories.

The infra components and applications on Staging clusters are controlled by the *main* branch of the respective repositories, whilst Production clusters are synchronised against the *production* branch. This permits the Flux image

automation controllers to automatically deploy updated Helm Charts in Staging whilst ensuring that protected branches (a Pull Request is needed to merge the content into the *production* branch) and appropriate release controls are applied to the Production environment. See [Protected branches](#) for further details.

You can also refer to the diagram within the [D1 Architecture Repositories Structure](#) section of the document.

Update Automation

Flux enables continuous deployment through image automation controllers so new application versions can be deployed to any environment. Within the D1 architecture, this is used to automate deployments to Staging clusters.

During the Staging cluster bootstrap, the `image-reflector-controller` and `image-automation-controller` are deployed (see [here](#)). It is not necessary to deploy the automation components to Production clusters. Pushes to the *main* branches will be performed by at least one Staging cluster.

These controllers work together to automatically update the `HelmReleases` stored in the *main* branch. More details on the process can be found [here](#).

This automation is built around different components:

1. `ImageRepository` CR. This tells Flux which container registry to scan for new tags
2. `ImagePolicy` CR. This tells Flux which [semver](#) range to use when filtering tags
3. Specific markers on the affected manifests to tell where Flux should update the version and with which policy (i.e. in a `HelmRelease.spec.chart.spec.version`)
4. `ImageUpdateAutomation` CR. This tells Flux which Git repository to write image updates to. The PAT associated with the `GitRepository` CR used by `ImageUpdateAutomation` must have enough privileges to perform pushes on the *main* branch (see [here](#))

As soon as Flux detects an update then:

5. The `HelmRelease` manifests will be updated on the *main* branch
6. The `HelmRelease` on all the Staging clusters will be reconciled to the new version

To update Production clusters:

7. Open and approve a pull request to the *production* branch
8. Wait for Flux to reconcile the updated **HelmRelease** CRs on all the Production clusters

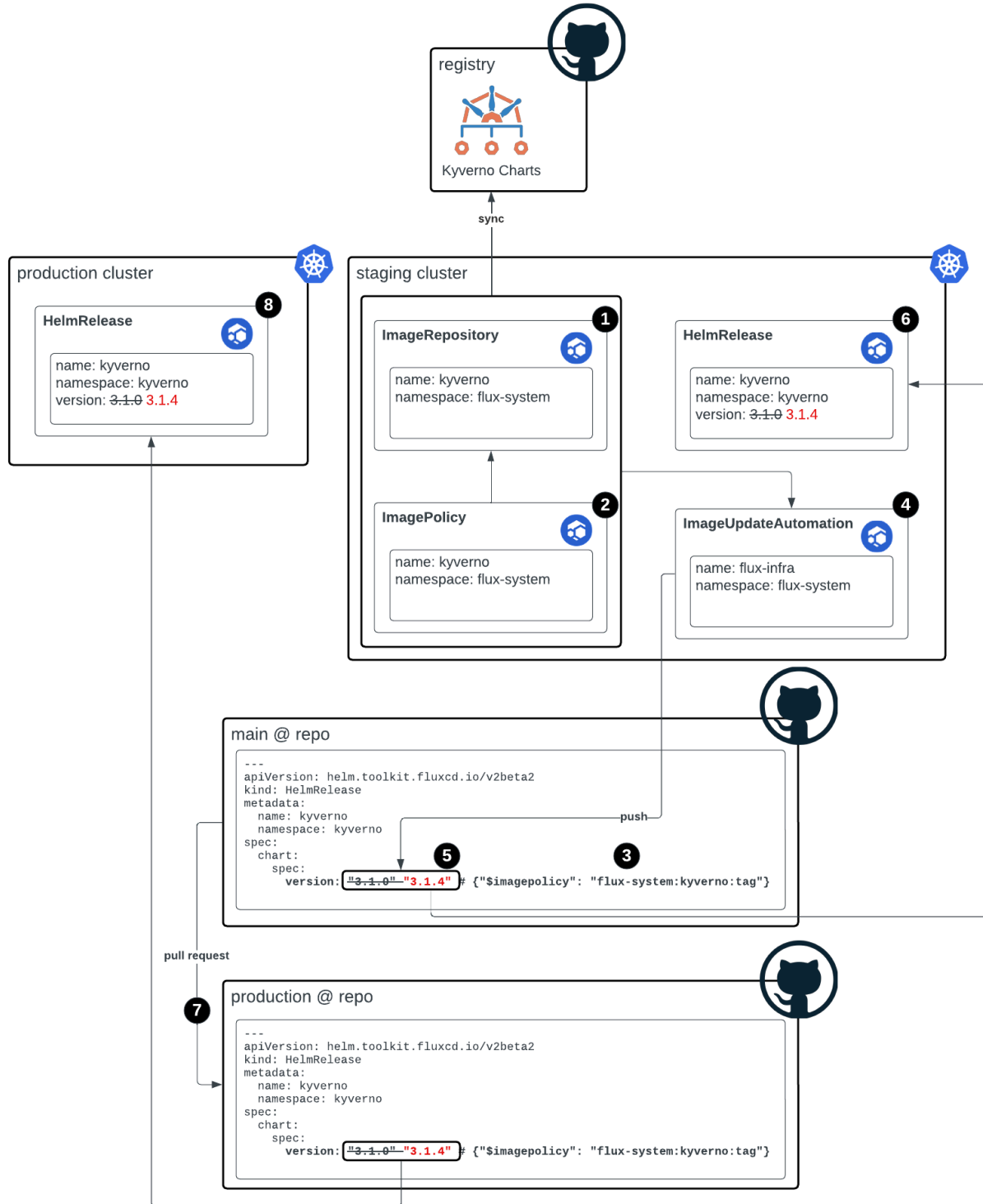


Figure 2: Update Automation

Following the d1 model, **ImageRepository**, **ImagePolicy** and manifest markers are pushed to the repository by the tenant responsible. **ImageUpdateAutomation** CR and the RBAC needed to reconcile the update automation components are deployed on the fleet repository by the Platform team for each tenant.

Example Workflows

Cluster Onboarding

The bootstrap procedure is a one-time operation that sets up the Flux controllers on the cluster (in this case a Staging cluster), and configures the delivery of platform components and applications.

The bootstrap procedure requires the following:

- Flux CLI
- A Github Bot Account Personal Access Token (PAT) with [suitable permissions](#)
- Cluster admin permissions on a provisioned Kubernetes cluster

The Flux CLI or Terraform Module can be called within a cluster provisioning pipeline or used from an administrator's machine. See the [flux bootstrap command](#) for further details, discussion on Flux CLI usage can be found in [Flux CLI usage & Alternatives](#).

The Flux CLI will use the bot PAT to push two commits to the *d1-fleet* repository:

1. The first commit is used to write the configuration of the *clusters/staging/flux-system/gotk-components.yaml* file, which contains the **flux-system Namespace**, RBAC, **NetworkPolicies**, CRDs and the controller **Deployments**
2. Second commit to create the *clusters/staging/flux-system/gotk-sync.yaml* file which contains the Flux **GitRepository** and **Kustomization** custom resources for setting up the cluster reconciliation

This Flux CLI will perform the following actions on the cluster:

3. Creates a Kubernetes **Secret** named **flux-system** in the flux-system namespace that contains the bot PAT
4. Builds the *cluster/staging/flux-system* kustomize overlays with the multi-tenancy patches and applies the generated manifests to the cluster to kick off the reconciliation

From this point on, the Flux controllers will:

5. Reconcile the cluster state with the desired state within the fleet repository
6. The tenant folder within the fleet repository contains **GitRepository** CRs that point to *d1-infra* and *d1-apps* repositories as further sources, as well as bootstrap configuration (namespace/RBAC) for those tenants

7. The Flux controllers subsequently reconcile the cluster state with the desired state within the *d1-infra* and *d1-apps* repositories

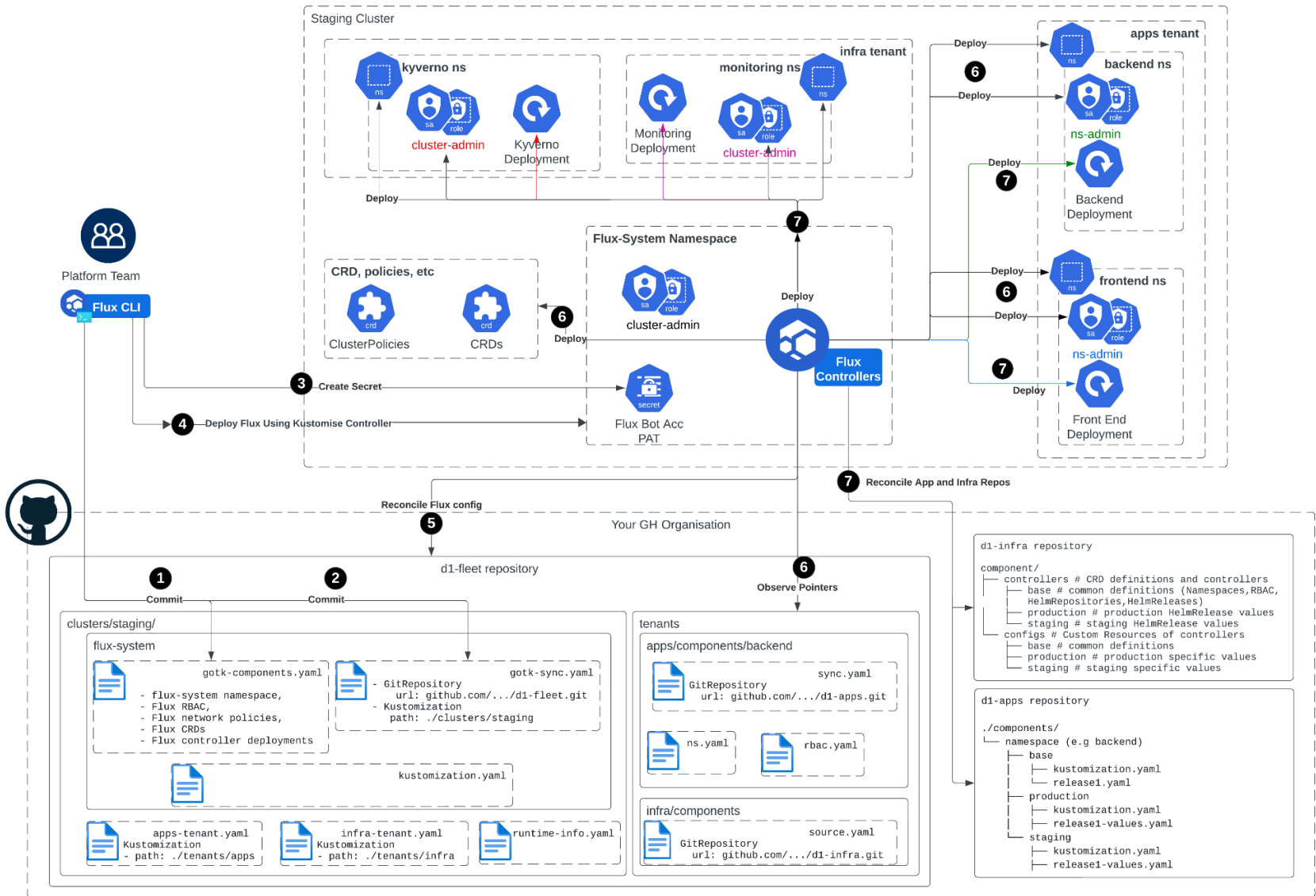


Figure 3: Example Workflow - Cluster onboarding

Adding a New Infra Component

This process is followed by the Platform team to add a new infrastructural component to the fleet. The diagram below shows where and when the manifests for a new infra component are pushed to the different repositories.

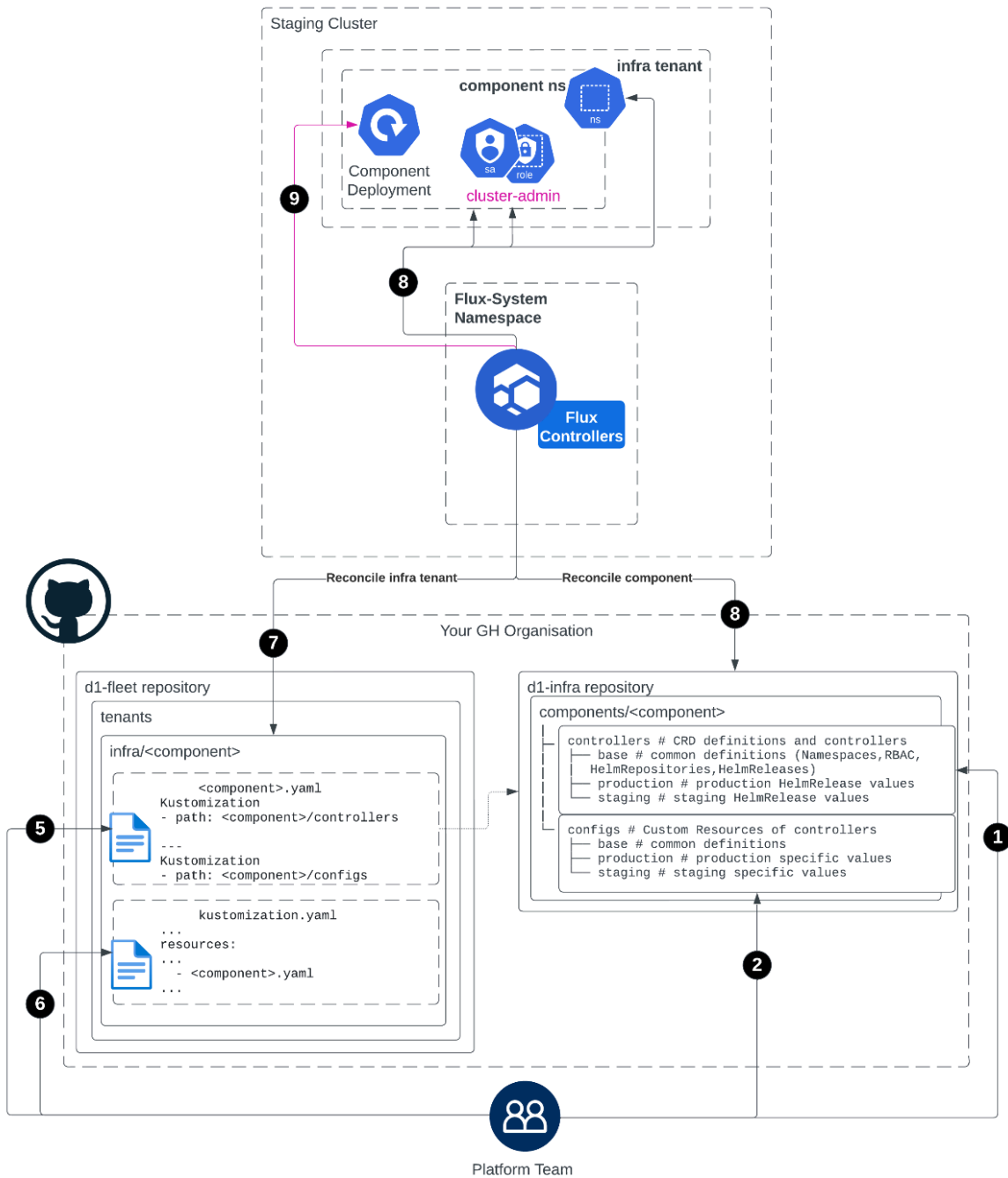


Figure 4: Example Workflow - Adding a new infrastructure component

To onboard a component from the *d1-infra* repository, the Platform team should fulfil the following requirements:

- Read-write access to *d1-fleet* repository
- Read-write access to *d1-infra* repository, on *main* and on *production* branches

The Platform team should then push on *d1-infra main* branch:

1. Add the infra component manifests and overlay to the `[d1-infra]/components/NEW_COMPONENT/controllers` folder, in order to
 - a. Deploy the needed **Namespace**
 - b. Deploy a new Flux **ServiceAccount** that is going to be responsible for the reconciliation of all the controllers belonging to that namespace
 - c. Deploy a new **ClusterRoleBinding** that will assign **cluster-admin ClusterRole** to the **Flux SA**
 - d. Deploy the component itself (i.e. using **HelmRelease**) and any needed CRDs
2. Add the needed manifests and overlay to the `[d1-infra]/components/NEW_COMPONENT/configs`, in order to deploy any configuration and custom resources used by that component
3. Automatize updates (optional):
 - a. Add **ImageRepository** and **ImagePolicy** CR to `[d1-infra]/update` folder and reference it in `[d1-infra]/kustomization.yaml` manifest
 - b. Add all the needed markers for the update automation workflow

Then, the team should push on *d1-fleet main* branch to start the reconciliation:

4. (Optional) add any useful cluster variable to `[d1-fleet]/clusters/CLUSTER/runtime-info.yaml` **ConfigMap**
5. Add new **Kustomizations** in `[d1-fleet]/tenants/infra/components` that will act as a pointer to the newly created manifests in **controllers** and **configs** folders of *d1-infra* repo. An explicit dependency between controllers and configs has to be declared, in order for the controller to be deployed before the configs
6. Add the reference to this **Kustomization** in `[d1-fleet]/tenants/infra/components/kustomization.yaml` in order to start the reconciliation for this component

Flux will then:

7. Reconcile the infra tenant and then notice two new **Kustomizations** for the component (for **controllers** and **configs**)
8. Reconcile the RBAC and namespace from `[d1-infra]/components/NEW_COMPONENT/controllers` folder (**HelmRepository**, **HelmRelease**, etc)

9. Reconcile the `HelmRepository`, `HelmRelease` from `[d1-infra]/components/NEW_COMPONENT/controllers` folder
10. Reconcile the objects from `[d1-infra]/components/NEW_COMPONENT/configs` folder

Finally, the Platform team will open and approve a pull request on the `d1-infra/production` branch to apply the changes to the Production cluster.

An infrastructural tenant might be organised in the following way:

```
Unset
component/
├─ controllers # CRD definitions and controllers
│  └─ base # common definitions (NS, RBAC, HelmRepositories, HelmReleases)
│  └─ production # production specific HelmRelease values
│  └─ staging # staging specific HelmRelease values
└─ configs # Custom Resources of controllers
    └─ base # common definitions
    └─ production # production specific values
    └─ staging # staging specific values
```

You can find a practical example in the [Appendix](#).

Adding a New Application Team/Tenant

The Platform team should be responsible for adding a new tenant for each Application team, following the process below.

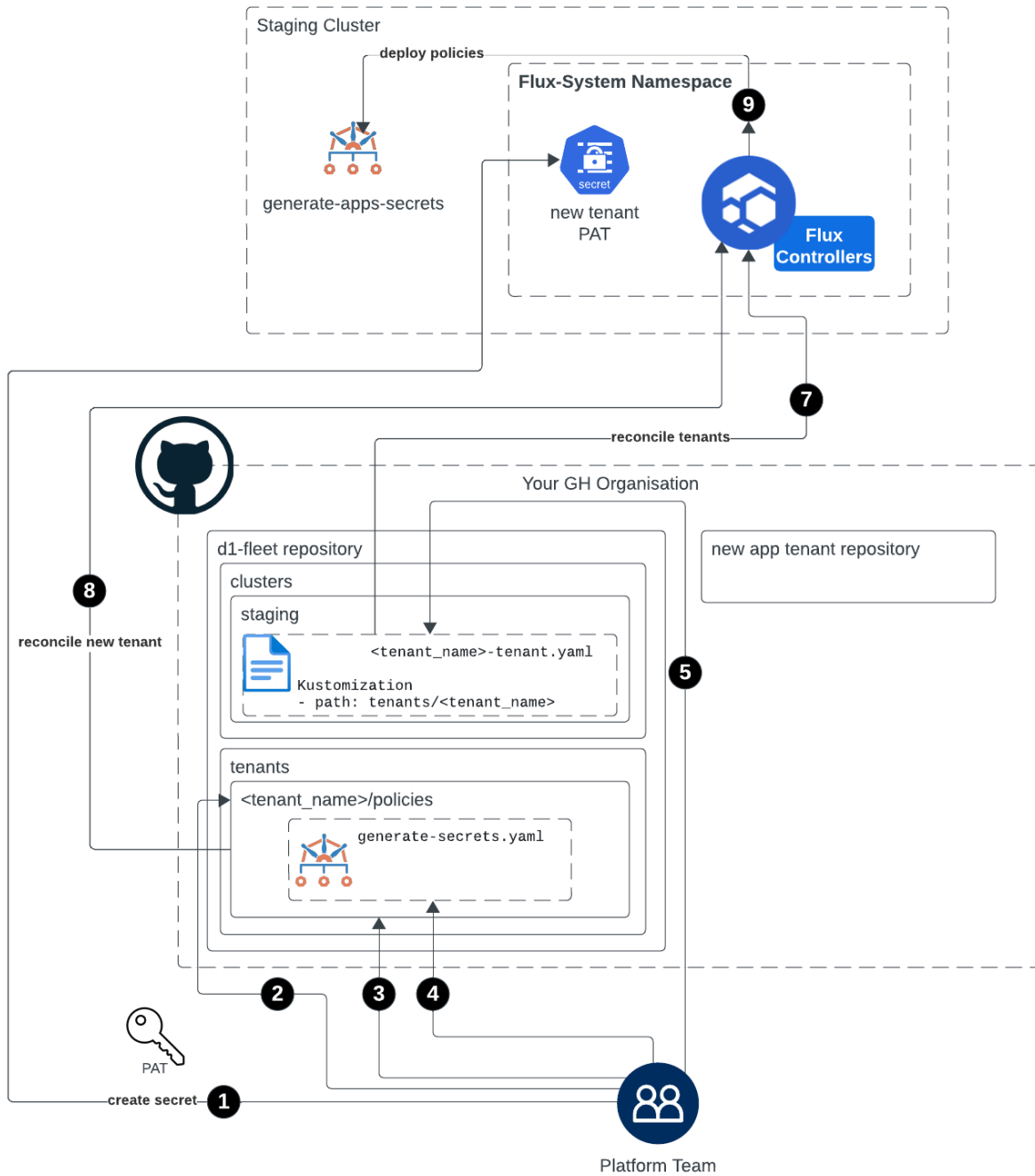


Figure 5: Example Workflow - Adding a new Application team/tenant

To onboard an application tenant the platform should fulfil the following requirements:

- Read-write access to *d1-fleet* repository
- Knows the PAT for the new tenant (see [here](#) for further considerations)

The Platform team should prepare the *d1-fleet* repo for the new tenant in the following way:

1. Create a secret storing the PAT for this tenant repo on all the affected clusters, via the `flux create secret git` command (see [here](#) for additional consideration on this topic)
2. Create a new folder `[d1-fleet]/tenants/NEW_TENANT` that will store all the tenant manifests (i.e. `[d1-fleet]/tenants/apps`). This will be populated by resources for each application (**Namespace**, **GitRepository**, **Kustomization**, **RBAC**, etc)
3. Make sure all the resources deployed by this tenant can be easily filtered by a label
4. Push a Kyverno policy in `[d1-fleet]/tenants/NEW_TENANT/policies` to sync all the secrets stored in **flux-system Namespace** belonging to this tenant to the application namespaces. Doing so will allow Flux controllers to be able to retrieve the needed secrets without crossing namespaces. This can be done using the previously declared label
5. Push a new **Kustomization** CR to each affected cluster in `[d1-fleet]/clusters/CLUSTER/NEW_TENANT-tenant.yaml` that will act as a pointer to the newly created folders and files
6. (optional) Push the manifests needed for the update automation:
 - a. Dedicated **Namespace apps-update**
 - b. **GitRepository** CR that will act as a pointer to the new tenant Git repository, with proper push permissions on the *main* branch
 - c. **ImageUpdateAutomation** CR that will update the manifests in the new tenant Git repository in the **components** folder
 - d. **RBAC (ServiceAccount + RoleBinding)** used to reconcile the update components for this new tenant
 - e. **Kustomization** CR that will start reconciliation for the **update** folder
 - f. Kustomize overlay to include all the resources declared before and to add the `toolkit.fluxcd.io/role: automation` label

Flux will then:

7. Reconcile the tenants and then notice a new **Kustomization** for the new tenant

8. Start reconciling the new tenant folder `[d1-fleet]/tenants/NEW_TENANT`
9. Reconcile the `ClusterPolicy` CRs

`Namespaces` and RBAC manifests for the tenant might be created using the [flux create tenant](#) command.

The Application team responsible for the new application tenant will have to manage their manifests inside the newly pointed Git repository.

An application tenant might be organised in namespaces in the following way:

```
Unset
.
├── components
│   ├── <namespace>
│   │   ├── base
│   │   ├── production
│   │   └── staging
│   └── <namespace>
│       ├── base
│       ├── production
│       └── staging
└── update
```

Managing an Application

Manifests that deploy namespaced objects belonging to the app are managed in the proper app tenant. Adding a new app will require joint work between the Platform team and the Application team responsible for that application. The Application team should identify the needed resources that can't be deployed by a namespaced admin role to the Platform team. This is because they cannot be automatically reconciled by the Flux role that will be assigned to the application's namespace

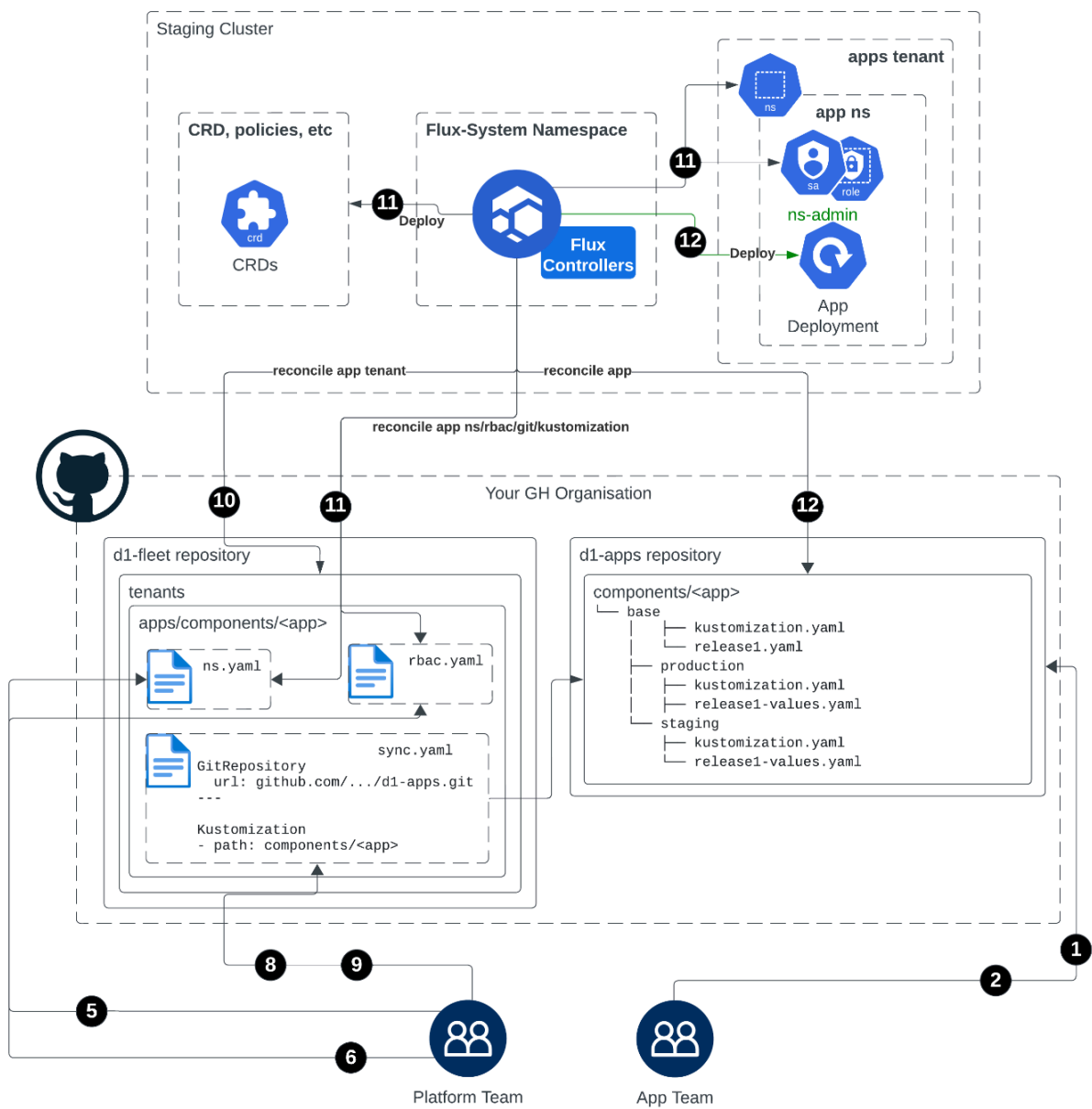


Figure 6: Example Workflow - Managing an application

To onboard a new application in an existing tenant the Platform team should fulfil the following requirement:

- Read-write access to *d1-fleet* repository

The Application team should fulfil the following requirement:

- Read-write access to the application tenant

The Application team should push on the application tenant:

1. Push the needed manifests (i.e. [HelmRepository](#) and [HelmRelease](#)) on their

- repository
- 2. Push any needed environment overlays
- 3. Pull request on *main* branch to apply changes on Production clusters
- 4. (optional) Push the manifests needed for the update automation (**ImagePolicy** and **ImageRepository**)

The Platform team should then push on *d1-fleet* repo all the manifests to prepare the environment creating non-namespaced objects:

- 5. **Namespace**
- 6. RBAC (**ServiceAccount** + **RoleBinding**) for Flux reconciliation for that specific application
- 7. **GitRepository** CR that will act as a pointer to the tenant Git repository
- 8. **Kustomization** CR to start the reconciliation for that repo on the application folder
- 9. Kustomize overlay to include all the resources declared before

Make sure all the resources above are tagged with a tenant identifier (see [here](#)).

Flux will then:

- 10. Reconcile the application tenant and then notice a new **Kustomization** for the application
- 11. Reconcile RBAC, CRDs and **Namespace**
- 12. Reconcile the objects from *[d1-apps]/components/APP* folder (**HelmRepository**, **HelmRelease**, etc)

This would be a possible structure for an application folder in *d1-fleet* repo:

```
Unset
./tenants/apps/components/<namespace>/
├─ kustomization.yaml
├─ namespace.yaml
├─ rbac.yaml
└─ sync.yaml
```


Further Implementation Guidance

Securing GitHub

The D1 reference architecture uses GitHub as the version control system. The next sections detail the security aspects to be taken into account for a safer Flux integration. For further information about GitHub security controls and how to implement them, please visit the [GitHub official documentation](#).

Having permissions to push configuration into the infra or fleet repositories is equivalent to having cluster-admin permissions over the cluster, as the account which is reconciling the state has been granted cluster-admin permissions. With the *d1-apps* repository, the blast radius would be reduced, as anything the applications team pushes to the *d1-apps* repository will be reconciled with namespace admin permissions. Hence, securing GitHub becomes critically important within Flux d1, or any GitOps architecture.

The security measures detailed below are overall guidance, but there may be additional considerations to be taken into account for each specific scenario.

GitHub as a Production Service

It is necessary to analyse the tolerance of the target organisation to a potential Flux outage. A GitHub outage would prevent the deployment of workloads within the onboarded clusters. The fallback would be to resort to manual deployment with administration of the cluster and management of applications conducted by the platform and Application teams respectively.

Therefore, analysing the service level agreements offered by GitHub in its different plans is a key factor to take into account within the whole Flux D1 architecture deployment. Internal Git repositories or mirrors may help to mitigate this risk.

User Access Control and Authorisation

In order to minimise the risk of malicious deployments, any interaction with Flux integrated repositories should only be allowed to those individuals and processes which actually need it. In order to guarantee proper access control to the Flux repositories the next GitHub controls should be enforced:

- Two-factor authentication: enforced for all users to prevent unauthorised access to GitHub repositories

- Team access controls to the repositories: proper segregation of duties and least privilege, using fine-grained access control to regulate access to the Flux repositories. Using GitHub Teams and adding the necessary users within the *platform* and application repositories allows granting only the necessary permissions
 - Limit access to *d1-fleet* and *d1-infra* repositories: access to these repositories should be reserved for Platform team members. Ideally, this GitHub team might be used to authorise direct administrative cluster access following a break glass process, in order to guarantee a 1 to 1 mapping between access to these repositories (that will be reconciled with cluster-admin roles) and those who can gain direct administrative cluster access (via *kubectl* or consoles)
 - Limit access to application repositories: repositories that store the manifests that will deploy the applications, i.e. *d1-apps*, should be reserved for Application team(s). Multiple teams will use multiple repositories that should be added as separate tenants

In Github, personal accounts have no possibility to create specific roles, but the other tiers permit the creation of up to three custom roles to allow more fine grained access control to the Flux repositories. Recommended permissions to be granted for the repositories and teams proposed above:

- Applications Team
 - *d1-fleet*: no permissions
 - *d1-infra*: no permissions
 - *d1-apps*: Members with write permissions, at least two admins within the repository or two owners
- Platform Team
 - *d1-fleet*: write permissions, at least two admins within the repository or two owners
 - *d1-infra*: write permissions, at least two admins within the repository or two owners
 - *d1-apps*: see [here](#) for different alternatives

This least privilege configuration may not be fully suitable for every organisation. If additional permissions are needed within the target organisation, it is always advisable to keep the access permissions to the minimum possible for both the Platform and Application team.

If it is necessary to segregate the permissions among different Application teams, it is recommended to add a different tenant.

Protected Branches

Considering the specific Flux integration with GitHub repositories, these controls are recommended at a branch level:

- *d1-fleet* Repo → *main* branch: Pull Requests to be approved by at least two approvers as Flux reconciles from this branch, and it is critical for cluster service account permissions
- *d1-infra* Repo → *main* branch: Pull Requests to be approved by at least one approver. This has the objective of having an extra review in case of overly-permissive manifests, and to detect earlier potential deviations. Flux bot has to be able to bypass this rule in order to automate updates on this branch. Further explanation on how to protect the branch and allow the bypass is given at the end of this section
- *d1-infra* Repo → *production* branch: Pull Requests to be approved by at least two approvers, as this is the branch Flux reconciles state and it will use cluster admin permissions, in the same way as the fleet repository
- *d1-apps* Repo → *main* branch: Pull Requests to be approved by at least one approver, to guarantee there is at least a peer review for any new artefact to be created within the cluster. Flux bot has to be able to bypass this rule in order to automate updates on this branch. Further explanation on how to protect the branch and allow the bypass is given at the end of this section
- *d1-apps* Repo → *production* branch: Pull Requests to be approved by at least two approvers. Similarly to what happens with the Flux infra repository, this is the branch the source controller will reconcile from and then deploy the new artefacts using an account with namespace administration permissions. Even though this service account does not have as wide permissions as the one used to reconcile the Infra or Fleet repositories, it is still possible to get control over cluster nodes deploying malicious artefacts and try to escalate privileges from there

In order to protect the branches previously mentioned, it is necessary to create a branch protection rule as next:

1. Go to the corresponding repository within GitHub
2. Click on Settings → branches → new protection rule
3. Check *require a pull request before merging*
 - a. Inside this rule, enable *require approvals*
 - b. Select the number of approvals suggested previously or any other suitable for your organisation
 - c. Select *Allow specified actors to bypass required pull requests*
 - d. Search for the flux-bot-account you created for the bootstrap and add it

4. Finally, populate the *Branch Name Pattern* at the top of the page with any pattern matching the needed criteria, in the previously exposed cases, you can populate with *main*
5. At the bottom of the page click on the *create* button

This will request every user who wants to make a change to the branch to require a pull request which also needs to be approved, but it will also allow the flux-bot-account to bypass the rule and push from the Flux deployed cluster directly to *main*, using the provisioned PAT.

Signed Commits

When developers work locally, Git allows them to set the author of their changes and the identity of the committer. This, potentially, makes it difficult for other people to be confident that commits and tags were actually created by the author in the commit message. To help solve this problem you can [sign commits](#) and tags. GitHub marks signed commits and tags with a verification status.

To give other users increased confidence in the identity attributed to each user's commits and tags, it is recommended that all the organisation's users are mandated to enable the *vigilant mode*. With *vigilant mode* enabled, all of the commits and tags are marked with their verification status.

Flux - Github Role Based Access Control (RBAC)

Bot Account

As mentioned in the [Cluster Onboarding section](#), a new GitHub account for the Flux bot is required. This account will be used by the Flux CLI and the Flux controllers running on clusters to authenticate with GitHub during cluster bootstrap, fleet (*d1-fleet* repo) and infrastructure (*d1-infra* repo) reconciliation.

For convenience, the newly created Flux bot account will be managed by the Platform team in your organisation with the following permissions:

- Read and write access to the *d1-fleet* repository (required for cluster bootstrap)
- Push access to the *main* branch of the *d1-fleet* repository (required for cluster bootstrap)
- Read and write access to the *d1-infra* and *d1-apps* repositories (required for cluster and application reconciliation and [image automation](#))

For these permissions to be granted, the bot account must be part of the GitHub Organisation and also part of an Organisation Team that has read and write permissions to all relevant repositories.

Personal Access Tokens

To allow Flux to use the bot account, Personal Access Tokens (PATs) must be generated, these are used by the Flux controllers to authenticate using the bot account. To prevent Application tenant privilege escalation through the use of a shared PAT, multiple PATs are generated, a single PAT for access to fleet and infra repositories, and a unique PAT for each new Application team with access limited to that Application team's repositories only.

The platform PAT has the following permissions for the *d1-infra* and *d1-fleet* repositories:

- *Administration* -> Access: Read-only
- *Commit statuses* -> Access: Read and write
- *Contents* -> Access: Read and write
- *Metadata* -> Access: Read-only

This token will be stored in all clusters to authenticate with GitHub to pull the fleet's desired state from the *d1-fleet* and *d1-infra* repositories. The token is also used to automate the Helm chart updates in the *d1-infra* repository, where the bot account has push access to the *main* branch.

To start reconciling the cluster from an application tenant, a new PAT has to be created with the following permissions:

- *Administration* -> Access: Read-only
- *Commit statuses* -> Access: Read and write
- *Contents* -> Access: Read and write
- *Metadata* -> Access: Read-only

This PAT should be limited in access to the application repository only (i.e. *d1-apps*).

This token will be stored in all clusters before onboarding the apps tenant (see [here](#)) and will be used by Flux to authenticate with GitHub and to start reconciling the tenant from the *d1-apps* repository. The token is also used to automate the Helm chart updates in the *d1-apps* repository, where the bot account has push access to the main branch (see [here](#) for additional considerations on read-write PAT).

As soon as the clusters need to be reconciled from a new tenant, a new dedicated PAT should be created, with limited access to the repositories belonging to that tenant.

N.b. Deploy Keys are an option supported by Flux, but are not considered within this guide owing to a lack of key expiry.

PAT Management Considerations

The above model is based for convenience on several trade-offs that must be carefully considered:

- PATs are generated by the same bot account that belongs to the Platform team with read-write permissions on all the repos. As an alternative, PATs might be generated from multiple accounts belonging to different teams with limited privileges only on the repositories the PAT will have access to. However, more granularity might require additional effort and less velocity
- PATs will be added as Kubernetes secrets by the Platform team during cluster bootstrap and during tenant onboarding. This means that the Platform team has to know these tokens. The Platform team might be the same team that manages the GitHub organisation, therefore they might be responsible for creating repositories, teams, bot accounts, and the PATs too. The team will then inherit read and write access to all the repositories, regardless of the tenant. This simplified model might not fit well with all organisations if a stricter segregation of duties is in place
- Mixed and more complex responsibility models affect how, when and by whom repositories, accounts, and PATs are created. As an example, credentials and/or PATs might be created by a dedicated GitHub team with administrative access to the organisation but without direct access to the clusters. Therefore, secrets — either bot credentials or ready-to-use PATs — will have to be securely shared between the GitHub and the Platform team, affecting how clusters are bootstrapped and/or tenants are onboarded
- As PATs expire by design, when deciding on a PAT management model, rotation must be taken into account. For a single Platform team-managed bot account, it should be possible for PATs to be rotated en masse, assisted by automation. Should separate teams be responsible for different bots and their respective PATs, then rotation will be their responsibility. No artefact will be pruned by Flux if the token is not renewed before the expiration date. However, no further update from the repo will be reconciled by the cluster until the PAT is renewed

Extending to other Git Providers

This model is specifically intended to work with GitHub permission models. This may be adapted to other Git providers as long as the PAT and team permissions are kept as similar as possible and follow the least privilege principle. Any consideration of SLA for the alternative provider is left to the enterprise.

Multi-Tenancy Considerations

Multi-Tenancy Configuration of Flux

Kustomization

The D1 model relies on the [Flux multi-tenancy configuration](#). As far as this model is concerned the tenants have restricted access to the cluster(s) according to the Kubernetes RBAC configured by the platform admins. The repositories owned by tenants are reconciled on the cluster(s) by Flux, under the Kubernetes account(s) assigned by platform admins.

According to the official documentation, the Flux installation is locked down using the following patches during the bootstrap process (see [here](#) for Production clusters Kustomize overlay):

- Deny cross-namespace access to Flux custom resources, thus ensuring that a tenant can't use another tenant's sources or subscribe to their events
- Deny access to Kustomize remote bases, thus ensuring all resources refer to local files, meaning only the Flux Sources can affect the cluster state
- All `Kustomizations` and `HelmReleases` which do not have `.spec.serviceAccountName` specified will use the default account from the tenant's namespace for deployment. Tenants have to specify a service account in their Flux resources to be able to deploy workloads in their namespaces as the default account has no permissions
- The `flux-system Kustomization` is set to reconcile under a service account with `cluster-admin` role, allowing platform admins to configure cluster-wide resources and provision the tenant's namespaces, service accounts and RBAC

Default Service Account

Even if application tenants are reconciled with service accounts with namespaced admin roles, particular attention should be paid to the `default` service account. `default` is the fallback account for `Kustomization` and `HelmReleases` with no `.spec.serviceAccountName` specified, therefore this should be carefully monitored to guarantee no unnecessary roles are assigned to it.

Flux Impersonation Model

In d1, Flux uses multiple service accounts to act on the cluster:

- `default` in `flux-system` is used as a fallback service account when no service account is explicitly specified in `.spec.serviceAccountName`
- `kustomize-controller` in `flux-system` namespace is used to reconcile `d1-fleet`, sync Flux components and reconcile the tenant pointers. In addition, this is used to create non-namespaced resources for the application tenant. This service account is used to reconcile the cluster policy that will sync needed secrets and configmaps across the namespaces
- `flux-infra` in each infrastructure namespace is used to reconcile the infrastructural controllers (details in Figure 6)
- `flux-apps` in each application namespace is used to reconcile the applicative components (details in Figure 7)

The multi-tenancy lock will allow Flux to only impersonate service accounts belonging to the same namespace where the `Kustomization/HelmRelease` is being deployed to.

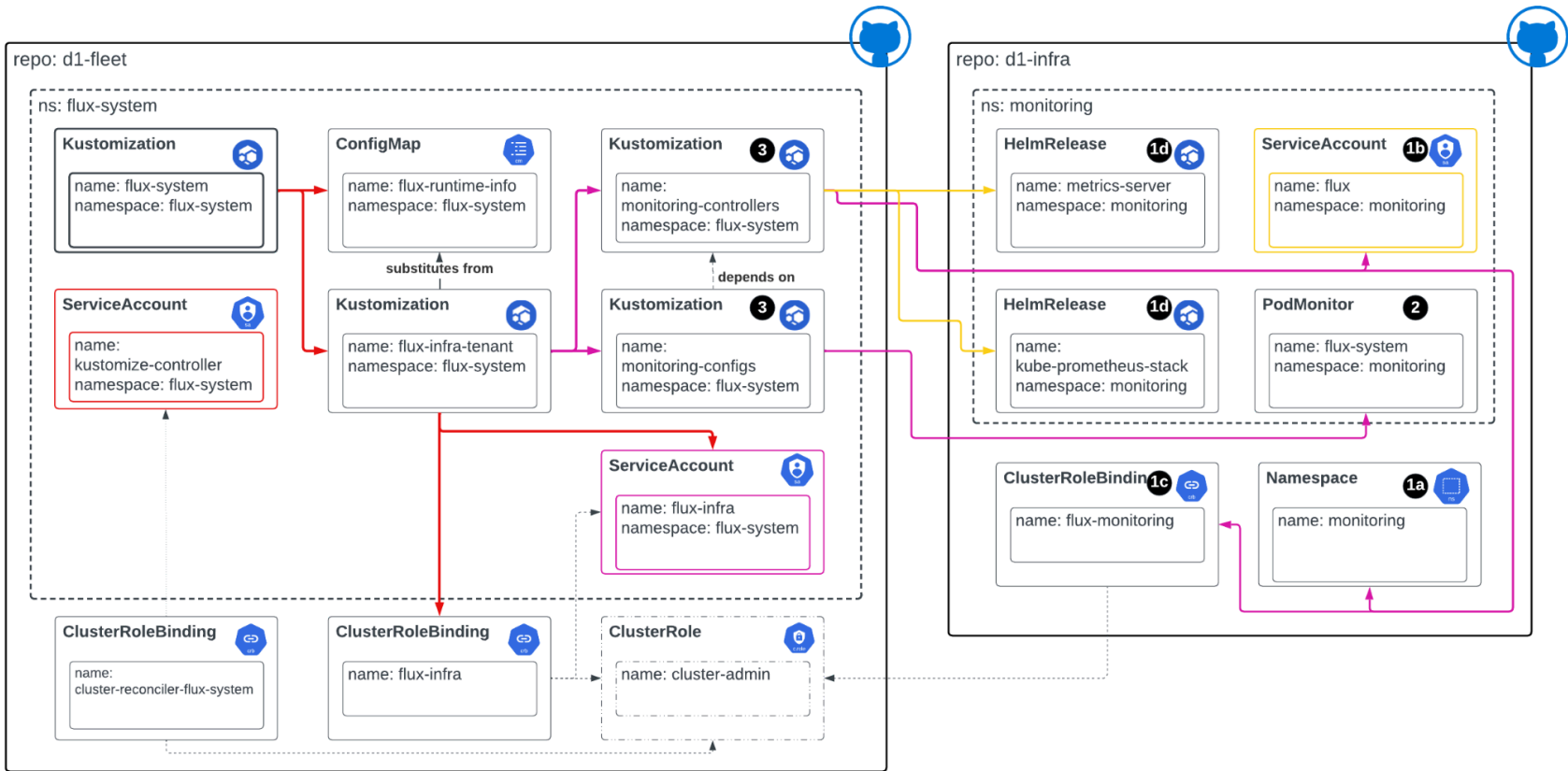


Figure 7: Flux impersonation model - flux-infra

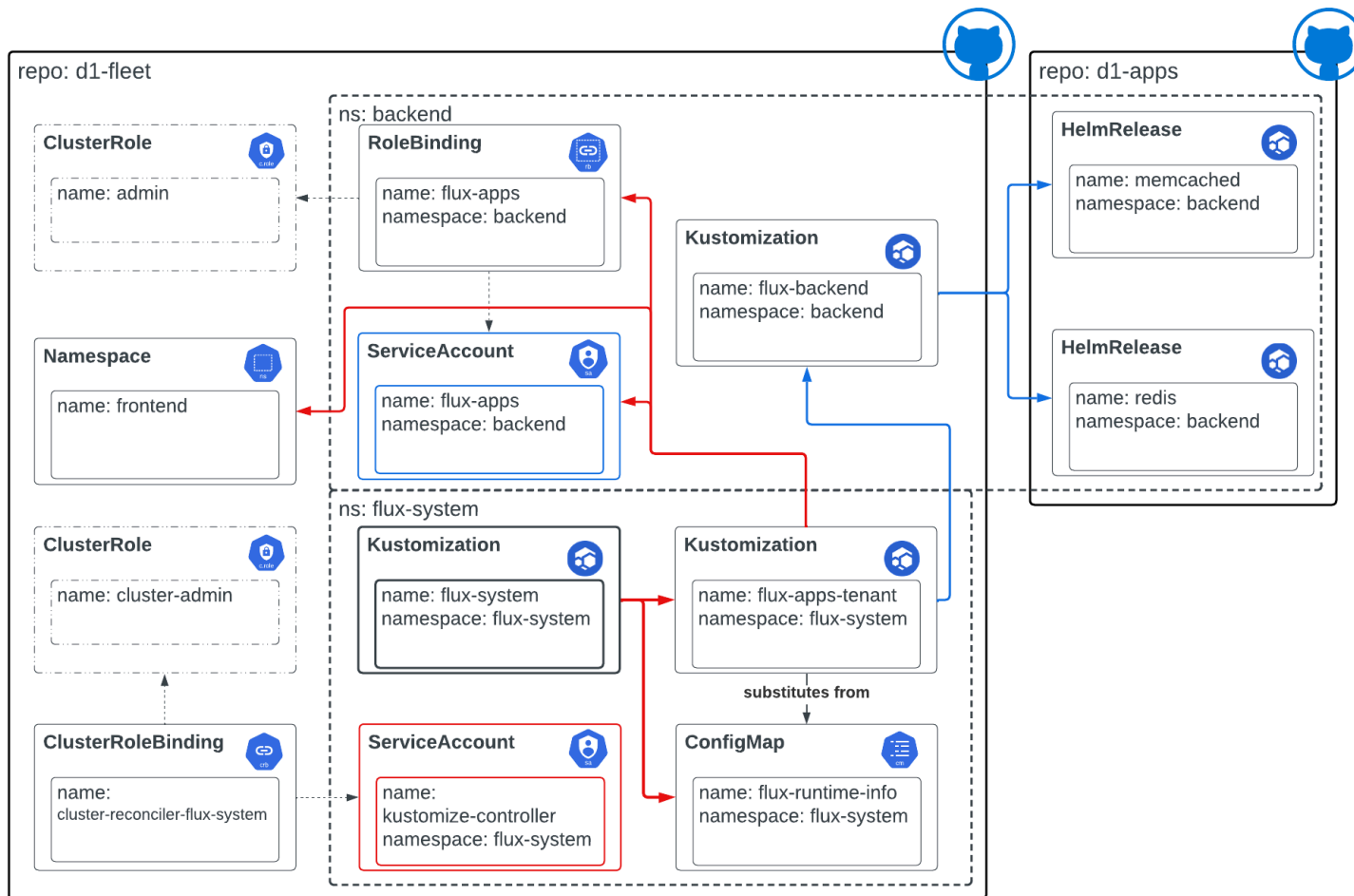


Figure 8: Flux impersonation model - flux-apps

Flux Controller Kubernetes RBAC Considerations for Multi-Tenancy

As part of onboarding an infrastructure component, or a namespace for an Application team to the cluster, the Platform team will define the permissions and service account used by the kustomize-controllers and helm-controllers to perform the deployments within Kubernetes. These are defined within the [d1-fleet repo](#) (example in link)

Within the provided example repos, it is important to note that these are namespace admin permissions for application tenants and cluster admin permissions for infrastructure components. You should analyse these permissions and define your own roles based on your use case with the following caveats:

- These permissions are not configured in line with least privilege
- If Infrastructure components are managed by external teams, providing those teams with cluster admin permissions via Flux provides an escalation route for non-Platform team members
- **admin ClusterRole** can vary from distribution to distribution. Binding to a role that includes '*'s is strongly discouraged
- It is likely that both application and Platform teams will need to configure their applications or infrastructure components to use custom resources such as cert-manager CRs, and these need to be added to the roles that Flux impersonates so that deployment can occur

Policy Pack for Multi-Tenancy

The following Kyverno policies are stored within the [\[d1-fleet\]/tenants/apps/policies](#). Each one of them has a specific purpose and might require additional consideration:

Secret Cloning and Least Privilege

Application tenants are able to access the Git repository via a secret stored in the application (i.e. backend/frontend) namespaces. The D1 model currently requires the Platform team to create these secrets (via `flux create secret git` command) in the flux-system namespace and then let a Kyverno policy sync them in the applicative namespaces belonging to that tenant ([here](#) for more details). The secret stores a single PAT with read-write permissions per tenant, for both reconciliation (fetch updates from the repo and apply them to the cluster) and update automation (update manifests on the *main* branch).

An attacker might be able to exploit a vulnerable application with enough privileges to access this read-write PAT and use it to affect the tenant *main* branch, directly affecting all the Staging clusters. As an alternative, an attacker might be able to

mount the secret in an arbitrary application, if no guardrails are implemented (i.e. admission controller policy).

It would be preferable to use a read-only PAT for reconciliation (synced to all the application namespace belonging to that tenant) and a read/write PAT for the update automation workflow (stored in a single namespace where no workflow is executed).

Runtime Information

`flux-runtime-info ConfigMap` can be used by `Kustomization` and therefore by workloads (infra or applicative). DI does not allow Flux controllers to make cross-namespace references. Similarly to PAT secrets, the `ConfigMap` is automatically copied in all the namespaces by Kyverno policy (`sync-flux-configmaps`), making the content of the `ConfigMap` potentially available to every workload.

No confidential information should be stored in this `ConfigMap`, in line with best practice.

Other Security and Availability Considerations

Flux CLI Usage & Alternatives

The guide describes cluster bootstrapping using the Flux CLI from a Platform team member's device, with the Flux CLI consuming the `kubeconfig` file on the device as well as the GitHub Flux Bot Personal Access Token.

With phishing and administrator device compromise becoming an increasingly common attack vector, devices should be hardened appropriately for this use case, with the Platform team subject to security awareness training.

Within regulated organisations, handling of privileged Kubernetes credentials, such as cluster-admin for Production clusters, would not normally be permissible on a Platform team member's device, including the Flux CLI. Alternatives include using a dedicated Bastion VM in the same VPC as the cluster to run the CLI or the [Flux Terraform provider](#), or using CI runners with credentials (PAT and Kubernetes credentials) stored as CI secrets.

Update Automation Service Level

In the D1 architecture, the controllers responsible for the update automation workflow are only deployed in Staging clusters (see [here](#)). That means that the manifests stored on the repo are automatically going to be updated only if at least one Staging cluster is available. This is something that should be taken into consideration and that would depend on which kind of service availability is expected for the update automation process.

Tenant Denial of Service Abuse Case

Flux's high-performance reconciliation has the capability to DOS the Kubernetes API endpoint if a source git repository is arbitrarily filled with a large number of Kubernetes resources.

This can occur regardless of any Kubernetes admission control policies as Flux controllers would need to pull, template the resources, authenticate, and send to the K8s API, which would then receive the resources and parse them before applying any mutating or validating policies.

In order to mitigate this potential problem, it is considered a good practice to set quotas for the total number of certain resources of all standard, namespaced resource types within the application namespaces by using an [object count quota](#).

[API Priority and Fairness \(APF\)](#) is another feature which mitigates a potential Denial of Service against the API Server. APF introduces a limited amount of queuing so that no requests are rejected in cases of very brief bursts. Requests are dispatched from queues using a fair queuing technique so that, for example, a poorly-behaved controller need not starve others (even at the same priority level).

About ControlPlane Enterprise for Flux CD

In January 2024, we launched [ControlPlane Enterprise for Flux CD](#) (CPE-Flux CD), designed to enhance highly regulated organisations that run and manage Production Kubernetes environments. CPE-Flux CD provides a secure, stable, and efficient subscription-based offering for enterprise Kubernetes environments and is designed to ensure the long-term sustainability of the upstream project.

We provide this sustainability by employing core Flux CD maintainers and offering additional security and vulnerability remediation enhancements. If your organisation requires or benefits from these enhancements over the upstream Flux CD project, CPE-Flux is an option to consider. It features:

- Enhanced security: additional guardrails, hardened and distroless container images, and FIPS-compliant Flux CD builds
- Vulnerability management: continuous scanning for CVEs, patching and distribution of Flux CD, and SLAs for critical vulnerabilities with signed VEX documentation
- Seamless upgrades and maintenance: zero-downtime migration, full upstream compatibility, OCI-compliant image repositories, compatibility with the latest six Kubernetes releases, and SLAs for critical vulnerability remediation
- Dedicated support: developer support portal for tracking and managing resolution and feature requests

This additional support, and enhanced security feature set, makes ControlPlane Enterprise for Flux CD a great choice for companies aiming for operational excellence of their Kubernetes environments, especially those dealing with the traits of demanding environments such as:

- Security-critical workloads requiring a clear segregation of duties
- Enforced least privilege Service Accounts
- At-scale deployments with large numbers of pods and nodes
- CI/CD pipelines with low latency delivery requirements
- Availability-critical deployments of new features
- Lack of highly specialised personnel capable of debugging low-level Flux CD integration issues
- Standards or regulations mandating that third-party integrations have to be compliant with security best practices

CPE-Flux CD is backed by ControlPlane's Technical Account Architect (TAA) offering and professional services team, providing additional specialised resources and support as needed.

It is important to emphasise that while CPE-Flux CD offers these additional security and remediation enhancements if they are not a requirement in your organisation you should use the upstream Flux CD project.

ControlPlane's team is available to support you in achieving secure, stable, and efficient Kubernetes Production environments, whether you choose to use CPE-Flux or the upstream Flux CD project. By offering CPE-Flux, ControlPlane aims to ensure the sustainability and continued development of the upstream Flux CD project.

Appendix

Administrative Tasks

Bootstrapping a Staging Cluster

Make sure to set the default context in your `kubeconfig` to your Staging cluster, then run bootstrap with:

```
Unset
export GITHUB_TOKEN=<Flux Bot PAT>

flux bootstrap github \
  --registry=ghcr.io/fluxcd \
  --components-extra=image-reflector-controller,image-automation-controller \
  --owner=<owner> \
  --repository=d1-fleet \
  --branch=main \
  --token-auth \
  --path=clusters/staging
```

This command will explicitly enable `image-reflector-controller` and `image-automation-controller`. See [here](#) for more details.

Bootstrapping a Production Cluster

Make sure to set the default context in your `kubeconfig` to the Production cluster you want to install Flux to, then run bootstrap with:

```
Unset
export GITHUB_TOKEN=<Flux platform PAT>

flux bootstrap github \
  --registry=ghcr.io/fluxcd \
  --owner=<owner> \
  --repository=d1-fleet \
  --branch=main \
  --token-auth \
  --path=clusters/prod-eu
```


Rotating the PATs

To rotate the main `flux-system Secret` (used to connect to the fleet and infra repository) on the currently selected cluster in `kubeconfig`, the Platform team should run this command:

```
Unset
flux create secret git flux-system \
  --namespace=flux-system \
  --url=https://github.com \
  --username=git \
  --password=$NEW_GITHUB_TOKEN
```

To rotate the secrets used by Flux to connect to the other tenant's repository on the currently selected cluster in `kubeconfig`, the Platform team should run this command:

```
Unset
flux create secret git flux-<tenant> \
  --namespace=flux-system \
  --url=https://github.com \
  --username=git \
  --password=$NEW_GITHUB_TOKEN
```

These commands should be executed for each of the affected clusters.

The [Kyverno policies](#) deployed by D1 will make sure that the secrets are propagated in the needed namespaces.

Adding an Infra Component

In this example, we are going to deploy MetalLB using Flux. [MetalLB](#) is “a load balancer implementation for bare metal [Kubernetes](#) clusters, using standard routing protocols”.

To do so, the Platform team should perform the following actions on fleet and infra repositories:

Infra repository

- Add `namespace.yaml` in `[d1-infra]/components/metallb/base`. This will contain:
 - metallb **Namespace** definition
 - **ServiceAccount** and **ClusterRoleBinding** for Flux reconciliation

```
Unset
apiVersion: v1
kind: Namespace
metadata:
  name: metallb
  labels:
    app.kubernetes.io/component: lb
    toolkit.fluxcd.io/tenant: platform-team
    pod-security.kubernetes.io/enforce: baseline
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  labels:
    app.kubernetes.io/component: lb
    toolkit.fluxcd.io/tenant: platform-team
  name: flux-lb
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: flux
  namespace: metallb
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: flux
  namespace: metallb
  labels:
    app.kubernetes.io/component: lb
    toolkit.fluxcd.io/tenant: platform-team
```

- Add `metallb.yaml` to `[d1-infra]/components/metallb/base`. This will contain **HelmRepository** and **HelmRelease**, with the common values used by all the environments

```

Unset
apiVersion: source.toolkit.fluxcd.io/v1beta2
kind: HelmRepository
metadata:
  name: metallb
  namespace: metallb
spec:
  interval: 12h
  url: https://metallb.github.io/metallb
---
apiVersion: helm.toolkit.fluxcd.io/v2beta2
kind: HelmRelease
metadata:
  name: metallb
  namespace: metallb
spec:
  serviceAccountName: flux
  interval: 1h
  chart:
    spec:
      version: "0.14.3"
      chart: metallb
  reconcileStrategy: ChartVersion
  sourceRef:
    kind: HelmRepository
    name: metallb
    interval: 12h
  install:
    crds: Create
    timeout: 9m
  upgrade:
    crds: CreateReplace
    timeout: 9m

```

- Add the IP pool for MetalLB as a cluster variable in *runtime-info.yaml*

```

Unset
apiVersion: v1
kind: ConfigMap
metadata:
  name: flux-runtime-info
  namespace: flux-system
  labels:
    toolkit.fluxcd.io/runtime: "true"
  annotations:

```

```

    kustomize.toolkit.fluxcd.io/ssa: "Merge"
data:
  ENVIRONMENT: "staging"
  GIT_BRANCH: "main"
  CLUSTER_NAME: "staging-1"
  CLUSTER_DOMAIN: "preview1.example.com"
  LB_IP_POOL: "192.168.10.0/24"

```

- Add the `IPAddressPool` and `BGPAdvertisement` CRs, in `[d1-infra]/components/metallb/configs`

```

Unset
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: main-pool
  namespace: metallb
spec:
  addresses:
  - ${LB_IP_POOL}
---
apiVersion: metallb.io/v1beta1
kind: BGPAdvertisement
metadata:
  name: bgpadvertisement-basic
  namespace: metallb
spec:
  ipAddressPools:
  - main-pool

```

- Add any needed environment overlays in `production` and `staging` folders, on both `controllers` and `configs`

Fleet Repository

- Add new `Kustomizations` in `[d1-fleet]/tenants/infra/components/lb.yaml`

```

Unset
tenants/infra
├── components
│   ├── admission.yaml
│   ├── kustomization.yaml
│   └── lb.yaml # <--

```

```
| |— monitoring.yaml
| |— rbac.yaml
| |— source.yaml
|— update
   |— automation.yaml
   |— kustomization.yaml
   |— sync.yaml
```

Unset

```
---
apiVersion: kustomize.toolkit.fluxcd.io/v1
kind: Kustomization
metadata:
  name: lb-controllers
spec:
  serviceAccountName: flux-infra
  interval: 1h
  retryInterval: 2m
  timeout: 10m
  prune: true
  wait: true
  sourceRef:
    kind: GitRepository
    name: flux-infra
    path: components/metallb/controllers/${ENVIRONMENT}
---
apiVersion: kustomize.toolkit.fluxcd.io/v1
kind: Kustomization
metadata:
  name: lb-configs
spec:
  serviceAccountName: flux-infra
  dependsOn:
    - name: lb-controllers
  interval: 1h
  retryInterval: 2m
  timeout: 5m
  prune: true
  wait: true
  sourceRef:
    kind: GitRepository
    name: flux-infra
```

```
path: components/metallb/configs/${ENVIRONMENT}
```

- Add a reference to *lb.yaml* in *[d1-fleet]/tenants/infra/components/kustomization.yaml* to kick start the reconciliation

```
Unset
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
namespace: flux-system
resources:
  - rbac.yaml
  - source.yaml
  - admission.yaml
  - monitoring.yaml
  - lb.yaml # <--
labels:
  - pairs:
      toolkit.fluxcd.io/tenant: infra
```

About

ControlPlane is a global cloud native and open source cybersecurity consultancy operating in London, New York, and Auckland. We have industry-leading expertise in the architecture, audit, and implementation of zero trust infrastructure for regulated industries. With a deep understanding of secure-by-design and secure-by-default cloud, Kubernetes, and supply chain security we conduct threat modelling, penetration testing, and cloud native security training to the highest standard.

ControlPlane is trusted as the partner of choice in securing: multinational banks; major public clouds; international financial institutions; critical national infrastructure programs; multinational oil and gas companies, healthcare and insurance providers; and global media firms.

<https://control-plane.io/>

Team

Andrea Martino
Miguel Ángel Hernández Ruiz
Rowan Baker

Reviewers

Andrew Martin
Eduardo Olarte
Stefan Prodan
Martin Stadler